Naming Conventions

1. Names representing packages should be in all lower case.

   Example: mypackage , com.company.application.ui

   The initial package name representing the domain name must be in lower case.

2. Names representing types must be nouns and written in mixed case starting with upper case.

   Example: Line , AudioSystem

3. Variable names must be in mixed case starting with lower case.

   Example: line , audioSystem

   Makes variables easy to distinguish from types, and effectively resolves potential naming collision as in the declaration "Line line";

4. Names representing constants (final variables) must be all uppercase using underscore to separate words.

   Example: MAX_ITERATIONS , COLOR_RED

   In general, the use of such constants should be minimized. In many cases implementing the value as a method is a better choice:

   Example:

```
int getMaxIterations()   // NOT: MAX_ITERATIONS = 25
{
        return 25;
}
```

This form is both easier to read, and it ensures a uniform interface towards class values.

6. Names representing methods must be verbs and written in mixed case starting with lower case.

   Example: getName() , computeTotalWidth()

   This is identical to variable names, but methods in Java are already distinguishable from variables

   by their specific form.

7. Abbreviations and acronyms should not be uppercase when used as name.

Example:

```
exportHtmlSource(); // NOT: exportHTMLSource();
openDvdPlayer(); // NOT: openDVDPlayer();
```

Using all uppercase for the base name will give conflicts with the naming conventions given above. A variable of this type whould have to be named dVD, hTML etc. which obviously is not very readable.

Another problem is illustrated in the examples above; When the name is connected to another, the readability is seriously reduced; The word following the acronym does not stand out as it should.

8. Private class variables should have underscore suffix.

Example:

```
class Person
{
        private String name_;
        ...
}
```

Apart from its name and its type, the scope of a variable is its most important feature. Indicating class scope by using underscore makes it easy to distinguish class variables from local scratch variables. This is important because class variables are considered to have higher significance than method variables, and should be treated with special care by the programmer.

A side effect of the underscore naming convention is that it nicely resolves the problem of finding reasonable variable names for setter methods:

Example:

```
void setName(String name)
{
    name_ = name;
}
```

An issue is whether the underscore should be added as a prefix or as a suffix. Both practices are commonly used, but the latter is recommended because it seem to best preserve the readability of

the name.

It should be noted that scope identification in variables have been a controversial issue for quite

some time. It seems, though, that this practice now is gaining acceptance and that it is becoming

more and more common as a convention in the professional development community.

9. Generic variables should have the same name as their type.

```
void setTopic(Topic topic)  // NOT: void setTopic(Topic value)
                            // NOT: void setTopic(Topic aTopic)
                          // NOT: void setTopic(Topic t)
void connect(Database database)  // NOT: void connect(Database db)

                                 // NOT: void connect(Database oracleDB)
```

Reduce complexity by reducing the number of terms and names used. Also makes it easy to

deduce the type given a variable name only.

If for some reason this convention doesn't seem to fit it is a strong indication that the type name is

badly chosen.

Non-generic variables have a role. These variables can often be named by combining role and type:

```
Point    startingPoint, centerPoint;
Name   loginName;
```

10. All names should be written in English.

English is the preferred language for international development.

11. Variables with a large scope should have long names, variables with a small scope can have short names .

Scratch variables used for temporary storage or indices are best kept short. A programmer

reading such variables should be able to assume that its value is not used outside a few lines of

code.

Common scratch variables :

For integers are i , j , k , m , n

For characters c and d.

12. The name of the object is implicit, and should be avoided in a method name.

Example: line.getLength(); // NOT: line.getLineLength();

The latter might seem natural in the class declaration, but proves superfluous in use, as shown in

the example.

13. The terms "get/set" must be used where an attribute is accessed directly.

Example:

```
employee.getName();
employee.setName(name);
matrix.getElement(2, 4);
matrix.setElement(2, 4, value);
```

14. "is" prefix should be used for boolean variables and methods.

Example:

isSet , isVisible , isFinished , isFound , isOpen

Using the "is" prefix solves a common problem of choosing bad boolean names like status or flag.

isStatus or isFlag simply doesn't fit, and the programmer is forced to chose more meaningful names.

Setter methods for boolean variables must have set prefix as in:

```
void setFound(boolean isFound);
```

There are a few alternatives to the is prefix that fits better in some situations. These are has, can

and should prefixes:

```
boolean hasLicense();
boolean canEvaluate();
boolean shouldAbort = false; ¥
```

15. The term compute can be used in methods where something is computed.

Example: valueSet.computeAverage();

matrix.computeInverse();

Give the reader the immediate clue that this is a potential time consuming operation, and if used

repeatedly, he might consider caching the result. Consistent use of the term enhances readability.

16. The term find can be used in methods where something is looked up.

Example:

vertex.findNearestVertex();

matrix.findSmallestElement();

node.findShortestPath(Node destinationNode);

Give the reader the immediate clue that this is a simple look up method with a minimum of

computations involved. Consistent use of the term enhances readability.

17. The term initialize can be used where an object or a concept is established.

Example: printer.initializeFontSet();

The American initialize should be preferred over the English initialise. Abbreviation init must be

avoided.

18. JFC (Java Swing) variables should be suffixed by the element type.

Example: widthScale , nameTextField , leftScrollbar , mainPanel , fileToggle , minLabel , printerDialog

Enhances readability since the name gives the user an immediate clue of the type of the

variable and thereby the available resources of the object.

19. Plural form should be used on names representing a collection of objects.

Example: Collection<Point> points;

int[] values;

Enhances readability since the name gives the user an immediate clue of the type of the variable and the operations that can be performed on its elements.

20. “ n” prefix should be used for variables representing a number of objects.

Example: nPoints , nLines

The notation is taken from mathematics where it is an established convention for indicating a

number of objects.

Note:

Sun use num prefix in the core Java packages for such variables. This is probably meant as an abbreviation of number of, but as it looks more like number it makes the variable name range and misleading. If ″number of″ is the preferred phrase, numberOf prefix can be used instead of just n. num prefix must not be used.

21. No suffix should be used for variables representing an entity number.

Example: tableNo , employeeNo

The notation is taken from mathematics where it is an established convention for indicating an

entity number.

An elegant alternative is to prefix such variables with an “i”: iTable, iEmployee. This effectively

makes them named iterators.

22. Iterator variables should be called i, j, k etc.

Example:

```
for (Iterator i = points.iterator(); i.hasNext(); )
{
    :
}
for (int i = 0; i < nTables; i++)
{
    :
}
```

The notation is taken from mathematics where it is an established convention for indicating iterators.

Variables named j, k etc. should be used for nested loops only.

23. Complement names must be used for complement entities.

Example: get/set , add/remove , create/destroy , start/stop , insert/delete , increment/decrement ,

old/new , begin/end , first/last , up/down , min/max , next/previous , old/new , open/close ,

show/hide , suspend/resume , etc.

Reduce complexity by symmetry.

24. Abbreviations in names should be avoided.

```
Example: computeAverage();  // NOT: compAvg();
 ActionEvent event;  // NOT: ActionEvent e;
 catch (Exception exception) {  // NOT: catch (Exception e) {
```

There are two types of words to consider. First are the common words listed in a language dictionary. These must never be abbreviated.

Never write:

cmd	instead of	command
comp	instead of	compute
cp	instead of	copy
e	instead of	exception
init	instead of	initialize
pt	instead of	point

etc.

Then there are domain specific phrases that are more naturally known through their acronym or abbreviations. These phrases should be kept abbreviated.

Never write:

HypertextMarkupLanguage	instead of	html
CentralProcessingUnit	instead of	cpu
PriceEarningRatio	instead of	pe

etc.

25. Negated boolean variable names must be avoided.

Example:

```
bool isError; // NOT: isNoError
bool isFound; // NOT: isNotFound
```

The problem arise when the logical not operator is used and double negative arises. It is not immediately apparent what “!isNotError” means.

26. Associated constants (final variables) should be prefixed by a common type name.

Example:

```
final int COLOR_RED = 1;
final int COLOR_GREEN = 2;
final int COLOR_BLUE = 3;
```

This indicates that the constants belong together, and what concept the constants represents.

An alternative to this approach is to put the constants inside an interface effectively prefixing their names with the name of the interface:

Example:

```
interface Color
{
      final int RED   = 1;
      final int GREEN = 2;
      final int BLUE  = 3;
}
```

27. Exception classes should be suffixed with Exception.

Example:

```
class AccessException extends Exception
{
          :
}
```

Exception classes are really not part of the main design of the program, and naming them like this

makes them stand out relative to the other classes.

28. Default interface implementations can be prefixed by Default.

Example:

```
class DefaultTableCellRenderer implements TableCellRenderer
{
        :
}
```

It is not uncommon to create a simplistic class implementation of an interface providing default

behaviour to the interface methods.

29. Singleton classes should return their sole instance through method getInstance.

Example:

```
class UnitManager
{
        private final static UnitManager instance_ = new UnitManager();
        private UnitManager()
        {
            ...
        }
        public static UnitManager getInstance() // NOT: get() or instance() or unitManager()          {
            return instance_;
        }
}
```

30. Classes that creates instances on behalf of others (factories) can do so through method new[ClassName]

Example:

```
class PointFactory
{
```

```
            public Point newPoint(...)
            {
                    ...
             }
       }
```

Indicates that the instance is created by new inside the factory method and that the construct is a

controlled replacement of new Point().

31. Functions (methods returning an object) should be named after what they return and procedures (void methods) after what they do.

Increase readability. Makes it clear what the unit should do and especially all the things it is not

supposed to do. This again makes it easier to keep the code clean of side effects.

32. Java source files should have the extension .java.

Example: `Point.java`

Enforced by the Java tools.

33. Classes should be declared in individual files with the file name matching the class name. Secondary private classes can be declared as inner classes and reside in the file of the class they belong to.

Enforced by the Java tools.

34. File content must be kept within 80 columns.

80 columns is the common dimension for editors, terminal emulators, printers and debuggers,

and files that are shared between several developers should keep within these constraints. It improves readability when unintentional line breaks are avoided when passing a file between programmers.

35. Special characters like TAB and page break must be avoided.

These characters are bound to cause problem for editors, printers, terminal emulators or debuggers when used in a multi-programmer, multi-platform environment.

36. The incompleteness of split lines must be made obvious.

Example:

```
totalSum = a + b + c + d + e;

method(param1, param2, param3);

setText ("Long line split" + "into two parts.");

for (int tableNo = 0; tableNo < nTables; tableNo += tableStep)

{

        ...

}
```

Split lines occurs when a statement exceed the 80 column limit given above. It is difficult to give

rigid rules for how lines should be split, but the examples above should give a general hint.

In general:

- Break after a comma.
- Break after an operator.
- Align the new line with the beginning of the expression on the previous line.

37. The package statement must be the first statement of the file. All files should belong to a specific package.

The package statement location is enforced by the Java language. Letting all files belong to an

actual (rather than the Java default) package enforces Java language object oriented

programming techniques.

38. The import statements must follow the package statement. import statements should be sorted with the most fundamental packages first, and grouped with associated packages together and one blank line between groups.

Example:

```
import java.io.IOException;

import java.net.URL;


import java.rmi.RmiServer;

import java.rmi.server.Server;
import javax.swing.JPanel;
import javax.swing.event.ActionEvent;

import org.linux.apache.server.SoapServer;
```

The import statement location is enforced by the Java language. The sorting makes it simple to browse the list when there are many imports, and it makes it easy to determine the dependiencies of the present package The grouping reduce complexity by collapsing related information into a common unit.

39. Imported classes should always be listed explicitly.

Example:

```
import java.util.List;  // NOT: import java.util.*;
import java.util.ArrayList;
import java.util.HashSet;
```

Importing classes explicitly gives an excellent documentation value for the class at hand and makes the class easier to comprehend and maintain.

Appropriate tools should be used in order to always keep the import list minimal and up to date.

40. Class and Interface declarations should be organized in the following manner:

- Class/Interface documentation.
- class or interface statement.
- Class (static) variables in the order public, protected, package (no access modifier),
- private.
- Instance variables in the order public, protected, package (no access modifier), private.
- Constructors.
- Methods (no specific order).

Reduce complexity by making the location of each class element predictable.

41. Method modifiers should be given in the following order:

<access> static abstract synchronized <unusual> final native

The <access> modifier (if present) must be the first modifier.

Example:

`public static double square(double a);` // NOT: static public double square(double a);

<access> is one of public, protected or private while <unusual> includes volatile and transient.

The most important lesson here is to keep the access modifier as the first modifier. Of the

possible modifiers, this is by far the most important, and it must stand out in the method

declaration. For the other modifiers, the order is less important, but it make sense to have a fixed

convention.

42. Type conversions must always be done explicitly. Never rely on implicit type conversion.

Example:

`floatValue = (int) intValue;` // NOT: floatValue = intValue;

By this, the programmer indicates that he is aware of the different types involved and that the

mix is intentional.

43. Array specifiers must be attached to the type not the variable.

Example:

`int[] a = new int[20];` // NOT: int a[] = new int[20]

The arrayness is a feature of the base type, not the variable.

44. Variables should be initialized where they are declared and they should be declared in the smallest scope possible.

This ensures that variables are valid at any time. Sometimes it is impossible to initialize a

variable to a valid value where it is declared. In these cases it should be left uninitialized rather

than initialized to some phony value.

45. Variables must never have dual meaning.

Enhances readability by ensuring all concepts are represented uniquely. Reduce chance of error by side effects.

46. Class variables should never be declared public.

The concept of Java information hiding and encapsulation is violated by public variables. Use private variables and access functions instead. One exception to this rule is when the class is essentially a data structure, with no behavior (equivalent to a C++ struct). In this case it is appropriate to make the class' instance variables public.

47. Arrays should be declared with their brackets next to the type.

Example:

```
double[] vertex; // NOT: double vertex[];
int[] count; // NOT: int count[];
public static void main(String[] arguments)
public double[] computeVertex()
```

The reason for is twofold. First, the array-ness is a feature of the class, not the variable. Second,

when returning an array from a method, it is not possible to have the brackets with other than the

type .

48. Variables should be kept alive for as short a time as possible.

Keeping the operations on a variable within a small scope, it is easier to control the effects and

side effects of the variable.

49. Only loop control statements must be included in the for() construction.

Example:

```
sum = 0;                        // NOT: for (i = 0, sum = 0; i < 100; i++)
for (i = 0; i < 100; i++)                   sum += value[i];
    sum += value[i];
```

Increase maintainability and readability. Make a clear distinction of what controls and what is contained in the loop.

50. Loop variables should be initialized immediately before the loop.

Example:

```
isDone = false;          // NOT: bool isDone = false;
while (!isDone)          //           :
{                        //         while (!isDone)
   :                     //         {
}                        //                    :
                         //         }
```

51. The use of do-while loops can be avoided.

do-while loops are less readable than ordinary while loops and for loops since the conditional is

at the bottom of the loop. The reader must scan the entire loop in order to understand the scope of

the loop.

In addition, do-while loops are not needed. Any do-while loop can easily be rewritten into a while

loop or a for loop. Reducing the number of constructs used enhance readbility.

52. The use of break and continue in loops should be avoided.

These statements should only be used if they prove to give higher readability than their

structured counterparts.

53. Complex conditional expressions must be avoided. Introduce temporary boolean variables instead.

Example:

```
bool isFinished = (elementNo < 0) || (elementNo > maxElement);
bool isRepeatedEntry = elementNo == lastElement;
if (isFinished || isRepeatedEntry)
{
   :
}
 // NOT:
if ((elementNo < 0) || (elementNo > maxElement)|| elementNo == lastElement)
{
      :
  }
```

By assigning boolean variables to expressions, the program gets automatic documentation. The
construction will be easier to read, debug and maintain.

54. The nominal case should be put in the if-part and the exception in the else-part of an if statement

Example:

```
boolean isOk = readFile(fileName);
if (isOk)
{
    :
}
else
{
    :
}
```

Makes sure that the exceptions does not obscure the normal path of execution. This is important
for both the readability and performance.

55. The conditional should be put on a separate line.

Example:

```
if (isDone)      // NOT: if (isDone) doCleanup();
doCleanup();
```

This is for debugging purposes. When writing on a single line, it is not apparent whether the test
is really true or not.

56. Executable statements in conditionals must be avoided.

Example:

```
InputStream stream = File.open(fileName, "w");
if (stream != null)
```

```
{
    :
}
// NOT: if (File.open(fileName, "w") != null))
  {
    :
  }
```

57. The use of magic numbers in the code should be avoided. Numbers other than 0 and 1can be considered declared as named constants instead.

```
private static final int TEAM_SIZE = 11;
:
Player[] players = new Player[TEAM_SIZE]; // NOT: Player[] players = new Player[11];
```

If the number does not have an obvious meaning by itself, the readability is enhanced by introducing a named constant instead.

58. Floating point constants should always be written with decimal point and at least one decimal.

Example:

```
double total = 0.0; // NOT: double total = 0;
double speed = 3.0e8; // NOT: double speed = 3e8;
double sum;
:
sum = (a + b) * 10.0;
```

This emphasize the different nature of integer and floating point numbers. Mathematically the two model completely different .

Also, as in the last example above, it emphasize the type of the assigned variable (sum) at a point

in the code where this might not be evident.

59. Floating point constants should always be written with a digit before the decimal point.

Example: `double total = 0.5; // NOT: double total = .5;`

The number and expression system in Java is borrowed from mathematics and one should adhere to mathematical conventions for syntax wherever possible. Also, 0.5 is a lot more readable than .

5; There is no way it can be mixed with the integer 5.

60. Static variables or methods must always be refered to through the class name and never through an instance variable.

Example: `Thread.sleep(1000);` // NOT: thread.sleep(1000);

This emphasize that the element references is static and independent of any particular instance.

For the same reason the class name should also be included when a variable or method is

accessed from within the same class.

61. Basic indentation should be 2.

Example:

```
for (i = 0; i < nElements; i++)
  a[i] = 0;
```

Indentation is used to emphasize the logical structure of the code. Indentation of 1 is to small to

acheive this. Indentation larger than 4 makes deeply nested code difficult to read and increase the

chance that the lines must be split. Choosing between indentation of 2, 3 and 4; 2 and 4 are the

more common, and 2 chosen to reduce the chance of splitting code lines.

Note:

The Sun recommendation on this point is 4.

62. Block layout should be as illustrated in example 1 below (recommended) or example 2, and must not be as shown in example 3. Class, Interface and method blocks should use the block layout of example 2.

Example:

1) `while (!done){`

```
  doSomething();
  done = moreToDo();
}
```

2) `while (!done)`

```
{
   doSomething();
   done = moreToDo();
}
```

3) while (!done)

```
    {
doSomething();
done = moreToDo();
                }
```

Example 3 introduce an extra indentation level which doesn't emphasize the logical structure

of the code as clearly as example 1 and 2.

63. The class and interface declarations should have the following form:

Example:

```
class Rectangle extends Shape
  implements Cloneable, Serializable
{
     ...
}
```

This follows from the general block rule above.

Note :

that it is common in the Java developer community to have the opening bracket at the end of the line of the class keyword. This is not recommended.

64. Method definitions should have the following form:

```
public void someMethod()
   throws SomeException
{
    ...
}
```

65. White Space

- Operators should be surrounded by a space character.
- Java reserved words should be followed by a white space.

- Commas should be followed by a white space.
- Colons should be surrounded by white space.
- Semicolons in for statements should be followed by a space character.

```
a = (b + c) * d; // NOT: a=(b+c)*d

while (true) {                    // NOT: while(true){
   ...
   doSomething(a, b, c, d);   // NOT: doSomething(a,b,c,d);

 case 100 :                       // NOT: case 100:

 for (i = 0; i < 10; i++) {     // NOT: for(i=0;i<10;i++){
   ...
```

Makes the individual components of the statements stand out and enhances readability.

66. Method names can be followed by a white space when it is followed by another name.

Example:

```
doSomething (currentFile);
```

Makes the individual names stand out. Enhances readability. When no name follows, the space

can be omitted (doSomething()) since there is no doubt about the name in this case.

An alternative to this approach is to require a space after the opening parenthesis. Those that

adhere to this standard usually also leave a space before the closing parentheses:

```
doSomething( currentFile );
```

This do make the individual names stand out as is the intention, but the space before the closing

parenthesis is rather artificial, and without this space the statement looks rather asymmetrical

(doSomething(currentFile);).

67. Logical units within a block should be separated by one blank line.

```
// Create a new identity matrix
```

```
   Matrix4x4 matrix = new Matrix4x4();

   // Precompute angles for efficiency
   double cosAngle = Math.cos(angle);
   double sinAngle = Math.sin(angle);

// Specify matrix as a rotation transformation
   matrix.setElement(1, 1, cosAngle);
   matrix.setElement(1, 2, sinAngle);
   matrix.setElement(2, 1, -sinAngle);
   matrix.setElement(2, 2, cosAngle);

  // Apply rotation
   transformation.multiply(matrix);
```

Enhances readability by introducing white space between logical units.

68. Methods should be separated by three blank lines.

By making the space larger than space within a method, the methods will stand out within the class.

69. Variables in declarations can be left aligned.

Example:

```
TextFile    file;
int         nPoints;
double      x, y;
```

Enhances readability.

70. Statements should be aligned wherever this enhances readability.

Example:

```
 if      (a == lowValue)       computeSomething();
else if (a == mediumValue)    computeSomethingElse();
else if (a == highValue)      computeSomethingElseYet();
```

```
    value = (potential          *  oilDensity)    /  constant1 +
            (depth              *  waterDensity) /  constant2 +
            (zCoordinateValue *  gasDensity)    /  constant3;

   minPosition      = computeDistance(min,      x, y, z);
   averagePosition = computeDistance(average, x, y, z);

  switch (phase) {
 case PHASE_OIL       :  text = "Oil";      break;
 case PHASE_WATER :  text = "Water";  break;
 case PHASE_GAS      :  text = "Gas";     break;
}
```

There are a number of places in the code where white space can be included to enhance readability even if this violates common guidelines. Many of these cases have to do with code alignment.

71. Tricky code should not be commented but rewritten.

In general, the use of comments should be minimized by making the code self-documenting by

appropriate name choices and an explicit logical structure.

72. Javadoc comments should have the following form:

Example:

```
/** *
Return lateral location of the specified position.
* If the position is unset, NaN is returned.
*
* @param x X coordinate of position.
* @param y Y coordinate of position.
* @param zone Zone of position.
* @return Lateral location.
* @throws IllegalArgumentException If zone is <= 0.
*/
 public double computeLocation(double x, double y, int zone) throws IllegalArgumentException
 {
          ...
}
```

A readable form is important because this type of documentation is typically read more often inside the code than it is as processed text.

Note :

- The opening /** on a separate line
- Subsequent * is aligned with the first one
- Space after each *
- Empty line between description and parameter section.
- Alignment of parameter descriptions.
- Punctuation behind each parameter description.
- No blank line bewteen the documentation block and the method/class.

Javadoc of class members can be specified on a single line as follows:

```
  /** Number of connections to this database */
  private int nConnections_;
```

73. There should be a space after the comment identifier.

```
// This is a comment NOT: //This is a comment
/**                              NOT: /**
 * This is a javadoc                  *This is a javadoc
 * comment                             *comment
 */                                      */
```

Improves readability by making the text stand out.

74. Use // for all non-JavaDoc comments, including multi-line comments.

```
// Comment spanning
 // more than one line.
```

Since multilevel Java commenting is not supported, using // comments ensure that it is always possible to comment out entire sections of a file using /* */ for debugging purposes etc.

75. Comments should be indented relative to their position in the code.

```
while (true) {                          // NOT: while (true) {
          // Do something                                  // Do something
           something();                                     something();
}                                                        }
```

This is to avoid that the comments break the logical structure of the program.

76. The declaration of anonymous collection variables should be followed by a comment stating the common type of the elements of the collection.

Example:

```
private Vector points_;                    // of Point
private Set       shapes_;                // of Shape
```

Without the extra comment it can be hard to figure out what the collection consist of, and thereby how to treat the elements of the collection. In methods taking collection variables as input, the common type of the elements should be given in the associated JavaDoc comment.

Whenever possible one should of course qualify the collection with the type to make the comment superflous:

```
  private Vector<Point>  points_;
  private Set<Shape>      shapes_;
```

77. All public classes and public and protected functions within public classes should be documented using the Java documentation (javadoc) conventions.

This makes it easy to keep up-to-date online code documentation.